Simplifying Algebraic Functional Systems

Kop, C.L.M.

published in
Lecture Notes in Computer Science
2011

DOI (link to publisher)
10.1007/978-3-642-21493-6_13

document version
Publisher's PDF, also known as Version of record

Simplifying Algebraic Functional Systems

Cynthia Kop

Department of Computer Science
VU University Amsterdam, The Netherlands

Abstract. A popular formalism of higher order rewriting, especially in the light of termination research, are the *Algebraic Functional Systems* (AFSs) defined by Jouannaud and Okada. However, the formalism is very permissive, which makes it hard to obtain results; consequently, techniques are often restricted to a subclass. In this paper we study termination-preserving transformations to make AFS-programs adhere to a number of standard properties. This makes it significantly easier to obtain general termination results.

Keywords: higher order term rewriting, algebraic functional systems, termination, transformations, currying, η-expansion.

1 Introduction

The last years have seen a rise in the interest in higher order rewriting, especially the field of termination. While this area is still far behind its first order counterpart, various techniques for proving termination have been developed, such as monotone algebras [11], path orderings [4,1] and dependency pairs [12,9,8]. Since 2010 the annual termination competition [13] has a higher order category.

However, a persistent problem is the lack of a fixed standard. There are several formalisms, each dealing with the higher order aspect in a different way, as well as variations and restrictions. Each style has different applications it models better, so it is hard to choose one over the others. Because of the differences, results in one formalism do not, in general, carry over to the next.

Consequently, when the topic of a higher order termination competition was brought up last year, the first question was: "What formalism?" Even having settled on monomorphic Algebraic Functional Systems, neither of the participating groups could immediately deal with the other's benchmarks, since one group used functional syntax and the other applicative.

In this paper we seek to alleviate this situation by studying transformations of Algebraic Functional Systems. AFSs, which were introduced as a modelling of functional programming languages, are an often recurring format in higher-order termination research, although most of the time they appear with some restrictions. Here we study an unrestricted version, with polymorphic types. After transforming an AFS it will satisfy a number of pleasant properties, such as β-normality and possibly η-normality, and we can freely swap between applicative and functional notation. The transformations are designed to have as little impact as possible and to preserve both termination and non-termination.

F. Winkler (Ed.): CAI 2011, LNCS 6742, pp. 201–215, 2011.

The aim of this work is to simplify termination research and tools for AFS. Without changing the syntactical freedom of the formalism, most unwelcome intricacies of the syntax can be eliminated in the input module of a tool. Techniques which are defined on a restricted subclass of AFSs (for example when rules are assumed to be η-normal) become generally applicable.

We will first present the definition of (polymorphic) AFSs. Section 3 sketches the problems we aim to solve; Sections 4–8 discuss various transformations. In Section 9 we will say a few words about generalising existing results.

2 Preliminaries

Algebraic Functional Systems (AFSs) were first defined in [3], but we follow (roughly) the more common definitions of [4]. Rather than using type declarations for variables in an environment, we annotate variables with their types directly in terms. This avoids the need to keep track of an environment.

Types. Given a set of *type constructors* $\mathcal{B}$, each with a fixed arity $ar(b)$, and a set of *type variables* $\mathcal{A}$, the set of *polymorphic types* is defined by the grammar:

$$\mathcal{T} = \alpha \mid b(\mathcal{T}^n) \mid \mathcal{T} \to \mathcal{T} \qquad (\alpha \in \mathcal{A},\ b \in \mathcal{B},\ ar(b) = n)$$

A *monomorphic* type does not contain type variables. We assume at least one type constructor has arity 0, so monomorphic types exist. A type $\sigma \to \tau$ is *functional*, and a type $b(\sigma_1, \ldots, \sigma_n)$ is a *data type*. Types are written as σ, τ, ρ, data types as ι, κ and type variables as $\alpha, \varepsilon, \omega$. The $\to$ operator associates to the right. A *type declaration* is an expression of the form $(\sigma_1 \times \ldots \times \sigma_n) \longrightarrow \tau$ with $\sigma_1, \ldots, \sigma_n, \tau \in \mathcal{T}$. A type declaration $() \longrightarrow \tau$ is usually just denoted τ. For any type σ, let $FTVar(\sigma)$ be the set of type variables occurring in σ.

Example 1. Examples of monomorphic types are `nat`, `nat` $\to$ `bool`, and `list` (`nat`), and an example of a non-monomorphic type is $\alpha \to$ `list`(α).

A *type substitution* θ is a finite mapping $[\alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n]$; $\mathrm{dom}(\theta) = \{\alpha_1, \ldots, \alpha_n\}$, and $\tau\theta$ is the result of replacing all α_i in τ by σ_i (with τ a type or type declaration). We say $\sigma \geq \tau$ if $\tau = \sigma\theta$ for some type substitution θ. For example, $\alpha \geq \alpha \geq \alpha \to \varepsilon \geq$ `nat` $\to$ `nat`, but not $\alpha \to \alpha \geq$ `nat` $\to$ `bool`. Substitutions θ, χ *unify* types σ, τ if $\sigma\theta = \tau\chi$[1]. We will use the following lemma:

Lemma 1 (most general unifiers). *If σ, τ are unifiable, there exist type substitutions θ, χ which unify σ, τ such that for any unifying substitutions θ', χ', there is a type substitution d such that $\theta'_{\restriction FTVar(\sigma)} = d \circ \theta$ and $\chi'_{\restriction FTVar(\tau)} = d \circ \chi$.*

The type substitutions θ, χ in this Lemma are called *most general unifiers*. The composition $d \circ \theta$ is defined as $[\alpha := d(\theta(\alpha)) | \alpha \in \mathrm{dom}(\theta)]$, and $\theta'_{\restriction FTVar(\sigma)}$ is defined as $[\alpha := \theta'(\alpha) | \alpha \in \mathrm{dom}(\theta') \cap FTVar(\sigma)]$.

[1] Note that this definition of unifiers considers the type variables in σ and τ as different entities: the types α and $b(\alpha)$ are unified by $\theta = [\alpha := b(\varepsilon)]$ and $\chi = [\alpha := \varepsilon]$.

Terms. Given a set $\mathcal{F}$ of *function symbols*, each equipped with a type declaration (notation f_σ), and an infinite set $\mathcal{V}$ of *variables*, the set of *terms* consists of those expressions for which we can derive $s : \sigma$ for some type σ, using the clauses:

(var)	$x_\sigma : \sigma$	if $x \in \mathcal{V}$ and σ a type
(fun)	$f_\sigma(s_1, \ldots, s_n) : \tau$	if $f_\rho \in \mathcal{F}$ and $\rho \geq \sigma = (\sigma_1 \times \ldots \times \sigma_n) \longrightarrow \tau$ and $s_1 : \sigma_1, \ldots, s_n : \sigma_n$
(abs)	$\lambda x_\sigma.s : \sigma \rightarrow \tau$	if $x \in \mathcal{V}$, σ a type and $s : \tau$
(app)	$s \cdot t : \tau$	if $s : \sigma \rightarrow \tau$ and $t : \sigma$

Moreover, variables must have a unique type in s: if for $x \in \mathcal{V}$ both x_σ and x_τ occur in s then $\sigma = \tau$. The abstraction operator λ binds occurrences of a variable as in the λ-calculus; term equality is modulo renaming of variables bound by an abstraction operator (α-conversion). Write $FVar(s)$ for the set of variables in s not bound by a λ. The $\cdot$ operator for application associates to the left. To maintain readability, we will regularly omit explicit type notation in function symbols and variables, and just assume the most general possible type.

Example 2. As a running example we will use the system $\mathcal{F}_{\mathtt{map}}$ with symbols:

$$\left\{ \begin{array}{llll} \mathtt{map}_{((\alpha\rightarrow\alpha)\times\mathtt{list}(\alpha))\longrightarrow\mathtt{list}(\alpha)}, & \mathtt{op}_{(\omega\rightarrow\varepsilon\times\alpha\rightarrow\omega)\longrightarrow\alpha\rightarrow\varepsilon}, & \mathtt{nil}_{\mathtt{list}(\alpha)}, & \mathtt{0}_{\mathtt{nat}}, \\ \mathtt{cons}_{(\alpha\times\mathtt{list}(\alpha))\longrightarrow\mathtt{list}(\alpha)}, & \mathtt{pow}_{(\alpha\rightarrow\alpha\times\mathtt{nat})\longrightarrow\alpha\rightarrow\alpha}, & \mathtt{s}_{(\mathtt{nat})\longrightarrow\mathtt{nat}} & \end{array} \right\}$$

An example term in this system is $\mathtt{map}(\lambda x.\mathtt{s}(x), \mathtt{cons}(\mathtt{0}, \mathtt{nil}))$. Since type annotations have been omitted, they should be imagined as general as possible to keep the term well-typed, so $\mathtt{cons}$, for instance, would be $\mathtt{cons}_{(\mathtt{nat}\times\mathtt{list}(\mathtt{nat}))\longrightarrow\mathtt{list}(\mathtt{nat})}$.

We extend type substitutions and $\geq$ to terms in the obvious way, with a type substitution θ replacing α by $\theta(\alpha)$ in all type denotations in the term.

Example 3. Using the symbols of Example 2, $\mathtt{op}_{(\alpha\rightarrow\varepsilon\times\varepsilon\rightarrow\alpha)\longrightarrow\varepsilon\rightarrow\varepsilon}(F_{\alpha\rightarrow\varepsilon}, G_{\varepsilon\rightarrow\alpha})[\alpha := \varepsilon, \varepsilon := \mathtt{nat}] = \mathtt{op}_{(\varepsilon\rightarrow\mathtt{nat}\times\mathtt{nat}\rightarrow\varepsilon)\longrightarrow\mathtt{nat}\rightarrow\mathtt{nat}}(F_{\varepsilon\rightarrow\mathtt{nat}}, G_{\mathtt{nat}\rightarrow\varepsilon})$.

A (term) substitution is the homomorphic extension of a mapping $[x^1_{\sigma_1} := s_1, \ldots, x^n_{\sigma_n} := s_n]$, where each $s_i : \sigma_i$. Substitutions are denoted $\gamma, \delta, \ldots$, the result of applying a substitution $s\gamma$. A substitution cannot affect bound variables; applying a substitution $(\lambda x_\sigma.s)\gamma$ assumes x occurs neither in domain nor range of γ (a safe assumption since we can rename bound variables). A *context* is a term C containing a special symbol $\Box_\sigma$. The result of replacing $\Box_\sigma$ in C by a term s of type σ is denoted $C[s]$. Here, s may contain variables bound by C.

β and η. $\Rightarrow_\beta$ is the monotonic relation generated by $(\lambda x.s) \cdot t \Rightarrow_\beta s[x := t]$. This relation is strongly normalising and has unique normal forms.

For a given set V of variables, we define restricted η-expansion: $C[s] \hookrightarrow_{\eta,V} C[\lambda x_\sigma.s \cdot x_\sigma]$ if $s : \sigma \rightarrow \tau$, x is fresh and the following conditions are satisfied:

1. s is neither an abstraction nor a variable in V
2. s in $C[s]$ is not the left part of an application.

By (2), s is not expanded if it occurs in a subterm of the form $s\ t_1 \cdots t_n$; (1) and (2) together guarantee that $\hookrightarrow_{\eta,V}$ terminates. Therefore every term s has a unique η, V*-long form* $s{\uparrow}^{\eta}_{V}$ which can be found by applying $\hookrightarrow_{\eta,V}$ until it is no longer possible. We say a term s is in η-long form if $s = s{\uparrow}^{\eta} = s{\uparrow}^{\eta}_{\emptyset}$.

Rules and Rewriting. An AFS consists of an alphabet $\mathcal{F}$ and a (finite or countably infinite) set $\mathcal{R}$ of *rules*. Rules are tuples $l \Rightarrow r$ where l and r are terms of the same type such that all variables and type variables in r also occur in l. The relation $\Rightarrow_{\mathcal{R}}$ induced by $\mathcal{R}$ is the monotonic relation generated by the β-rule and: $l\theta\gamma \Rightarrow_{\mathcal{R}} r\theta\gamma$ if $l \Rightarrow r \in \mathcal{R}$, θ is a type substitution and γ a substitution.

Example 4. Using the symbols $\mathcal{F}_{\mathtt{map}}$ from Example 2, let $R_{\mathtt{map}}$ be the set:

$$\left\{\begin{array}{ll} \mathtt{map}(F, \mathtt{nil}) & \Rightarrow \mathtt{nil} \\ \mathtt{map}(F, \mathtt{cons}(x, y)) & \Rightarrow \mathtt{cons}(F \cdot x, \mathtt{map}(F, y)) \\ \mathtt{pow}(F, 0) & \Rightarrow \lambda x.x \\ \mathtt{pow}(F, \mathtt{s}(x)) & \Rightarrow \mathtt{op}(F, \mathtt{exp}(F, x)) \\ \mathtt{op}(F, G) \cdot x & \Rightarrow F \cdot (G \cdot x) \\ \lambda x.F \cdot x & \Rightarrow F \end{array}\right\}$$

As before, types should be imagined as general as possible. The last rule, for example, should be read as $\lambda x_\alpha.F_{\alpha\to\epsilon} \cdot x_\alpha \Rightarrow F_{\alpha\to\epsilon}$. An example reduction:

$$\begin{array}{ll} \mathtt{map}(\underline{\mathtt{pow}(\lambda x.\mathtt{s}(\mathtt{s}(x)), \mathtt{s}(0))}, \mathtt{cons}(0, \mathtt{nil})) & \Rightarrow_{\mathcal{R}} \\ \underline{\mathtt{map}(\mathtt{op}(\lambda x.\mathtt{s}(\mathtt{s}(x)), \mathtt{pow}(\lambda x.\mathtt{s}(\mathtt{s}(x)), 0)), \mathtt{cons}(0, \mathtt{nil}))} & \Rightarrow_{\mathcal{R}} \\ \mathtt{cons}(\mathtt{op}(\lambda x.\mathtt{s}(\mathtt{s}(x)), \mathtt{pow}(\lambda x.\mathtt{s}(\mathtt{s}(x)), 0)) \cdot 0, \underline{\mathtt{map}(\ldots, \mathtt{nil})}) & \Rightarrow_{\mathcal{R}} \\ \mathtt{cons}(\mathtt{op}(\lambda x.\mathtt{s}(\mathtt{s}(x)), \underline{\mathtt{pow}(\lambda x.\mathtt{s}(\mathtt{s}(x)), 0)}) \cdot 0, \mathtt{nil}) & \Rightarrow_{\mathcal{R}} \\ \mathtt{cons}(\underline{\mathtt{op}(\lambda x.\mathtt{s}(\mathtt{s}(x)), \lambda y.y)} \cdot 0, \mathtt{nil}) & \Rightarrow_{\mathcal{R}} \\ \mathtt{cons}((\lambda x.\mathtt{s}(\mathtt{s}(x))) \cdot (\underline{(\lambda y.y) \cdot 0}), \mathtt{nil}) & \Rightarrow_{\beta} \\ \mathtt{cons}(\underline{(\lambda x.\mathtt{s}(\mathtt{s}(x))) \cdot 0}, \mathtt{nil}) & \Rightarrow_{\beta} \\ \mathtt{cons}(\mathtt{s}(\mathtt{s}(0)), \mathtt{nil}) & \end{array}$$

Note that the $\Rightarrow_\beta$ steps in this are also $\Rightarrow_{\mathcal{R}}$ steps since $\Rightarrow_\beta$ is included in $\Rightarrow_{\mathcal{R}}$ by definition; they are named separately for clarity.

3 Problems

The permissive nature of AFS-syntax makes it difficult to obtain general results. The first issue is the status of application. When extending first order results it is convenient to consider the $\cdot$ operator as a (polymorphic) binary function symbol. But this doesn't work very well with applicative systems, which have rules like $\mathtt{map} \cdot F \cdot (\mathtt{cons} \cdot x \cdot y) \Rightarrow \mathtt{cons} \cdot (F \cdot x) \cdot (\mathtt{map} \cdot F \cdot y)$; AFSs generated from functional programs in e.g. Haskell will commonly have such a form. Due to the repeated occurrence of the $\cdot$ symbol, no version of a higher-order path ordering [4,1] can handle this system. To avoid this problem, we might consider

$\cdot$ as a stronger construction, much like function application; this is done in Nipkow's *Higher-Order Rewriting Systems* (HRSs) [10], where terms are built using only abstraction and application. Ideally, a term $\mathtt{map} \cdot x \cdot (\mathtt{cons} \cdot y \cdot z)$ could be translated to its functional counterpart $\mathtt{map}(x, \mathtt{cons}(y, z))$. But initially this is impossible: a system with the rule $x \cdot 0 \Rightarrow f \cdot 0$ admits a self-loop $f \cdot 0 \Rightarrow f \cdot 0$, whereas the corresponding functional rule, $x \cdot 0 \Rightarrow f(0)$, is terminating.

Another difficulty is the form of the left-hand side of a rule. Methods like the dependency pair approach crucially rely on rules having a form $f(\ldots) \Rightarrow r$ or, in the recent dependency pair analysis for AFSs in [8], $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m \Rightarrow r$. Consequently, systems with rules like $\lambda x.F \cdot x \Rightarrow F$ cannot be handled.

Termination techniques are often defined only on a restricted subset of AFSs. Since most common examples are expressed in a well-behaved manner, this does not seem too high a price. However, a transformational approach, where for instance a term $f(s, t)$ is replaced by $t \cdot s$, is likely to create rules which do not follow the usual assumptions. Instead, we will see how any AFS can be transformed so that all rules have a form $l = f(\boldsymbol{s}) \cdot \boldsymbol{t} \Rightarrow r$ with l and r both β-normal and l not containing leading free variables. We tackle standard assumptions (monomorphism and η-normality) which can be made about terms, and show that, after the first transformations, functional and applicative syntax are interchangeable. We aim to keep the complexity of the transformations minimal: a finite system remains finite after transforming, a monomorphic system remains monomorphic.

4 Polymorphism

In a first step towards simplifying the system, let us investigate polymorphism. To start, observe that polymorphism is only needed to define rules, not terms.

Theorem 1. *If a system is non-terminating, there is an infinite reduction on monomorphic terms.*

Proof. Given an infinite reduction $s_0 \Rightarrow_{\mathcal{R}} s_1 \Rightarrow_{\mathcal{R}} \ldots$, let θ be a type substitution which replaces all type variables in s_0 by a type constructor b of arity 0. Since $\Rightarrow_{\mathcal{R}}$ does not create type variables and is closed under type substitution, $s_0\theta \Rightarrow_{\mathcal{R}} s_1\theta \Rightarrow_{\mathcal{R}} \ldots$ is an infinite monomorphic reduction.

Polymorphism has its purpose in defining rules: any set of rules corresponds with a monomorphic set, but instantiating type variables leads to infinitely many rules. Finiteness is a high price to pay, since both humans and computers have trouble with the infinite. Nevertheless, from a perspective of reasoning we might as well use monomorphic rules, as long as we remember how they were generated.

Let a *rule scheme* be a pair $l \Rightarrow r$ of equal-typed (polymorphic) terms, such that all free variables and type variables of r also occur in l. Given a set R of rule schemes, let $\mathcal{R}_R = \{l\theta \Rightarrow r\theta | l \Rightarrow r \in R$ and θ a type substitution mapping all type variables in l to monomorphic types$\}$. The following is evident:

Theorem 2. *For a given set of rule schemes R, the set $\mathcal{R}_R$ is a set of monomorphic rules and $\Rightarrow_R$ is terminating if and only if $\Rightarrow_{\mathcal{R}_R}$ is.*

Henceforth, *rules* are understood to be monomorphic. *Rule schemes* may not be. $\mathcal{R}$ indicates a set of rules, R a set of rule schemes.

A pleasant consequence of using monomorphic rules is that type substitution is no longer needed to define the rewrite relation $\Rightarrow_{\mathcal{R}}$; $s \Rightarrow_{\mathcal{R}} t$ if either $s \Rightarrow_\beta t$ or $s = C[l\gamma]$ and $t = C[r\gamma]$ for some substitution γ, context C and rule $l \Rightarrow r$.

In the following sections we will define transformations on a set of rule schemes. Note that a set $\mathcal{R}$ of rules is always generated from a set of rule schemes, since even for monomorphic systems $R := \mathcal{R}$ is a suitable set.

5 Leading Variables

The presence of leading variables in the left-hand side of a rule $l \Rightarrow r$ (that is, subterms $x \cdot s$ where x is free in l) hinders techniques like dependency pairs[2] and makes it impossible to swap freely between functional and applicative notation (see also Section 7). We can avoid this problem by making application a function symbol: replace $s \cdot t$ everywhere by $@_{(\sigma\to\tau\times\sigma)\longrightarrow\tau}(s,t)$ and add $@_{(\alpha\to\varepsilon\times\alpha)\longrightarrow\tau}(x,y) \Rightarrow x \cdot y$ to R. The resulting system is terminating if and only if the original was. However, as discussed in Section 3, this transformation is not very good. A mostly applicative system would become almost impossible to handle with conventional techniques. In addition, the new rule scheme uses type variables, while the original system might be monomorphic. Thus, we will use a more complicated transformation that leads to an easier system.

Sketch of the Transformation. We sketch the idea for transforming a monomorphic system. Polymorphism brings in additional complications, but the rough idea is the same. First we instantiate headmost variables with functional terms: for any rule $l = C[x \cdot s] \Rightarrow r$ all possible rules $l[x := f(\boldsymbol{y}) \cdot \boldsymbol{z}] \Rightarrow r[x := f(\boldsymbol{y}) \cdot \boldsymbol{z}]$ are added. Now when a rule with leading variables is used, we can assume these variables are not instantiated with a functional term. Second, we introduce a symbol @ and replace occurrences $s \cdot t$ in any rule by $@(s,t)$ if s is not functional, and its type corresponds with the type of a leading variable in any left-hand side. We add rules $@_\sigma(x,y) \Rightarrow x \cdot y$ only for those $@_\sigma$ occurring in the changed rules. With this transformation, the applicative `map` rule $\mathtt{map}_{(\mathtt{nat}\to\mathtt{nat})\to\mathtt{list(nat)}\to\mathtt{list(nat)}} \cdot F \cdot (\mathtt{cons} \cdot x \cdot y) \Rightarrow \mathtt{cons} \cdot (F \cdot x) \cdot (\mathtt{map} \cdot F \cdot y)$ either stays unchanged (if there are no rules with a leading variable of type $\mathtt{nat}\to\mathtt{nat}$ in the left-hand side) or becomes $\mathtt{map} \cdot F \cdot (\mathtt{cons} \cdot x \cdot y) \Rightarrow \mathtt{cons} \cdot @(F,x) \cdot (\mathtt{map} \cdot F \cdot y)$ (if there are).

5.1 Output Arity

Polymorphism complicates this transformation. Even with finite $\mathcal{F}$ there may be infinitely many terms of the form $f(\boldsymbol{x}) \cdot \boldsymbol{y}$. So assign to every $f_\sigma \in \mathcal{F}$ an integer $oa(f) \geq 0$; terms of the form $f(\boldsymbol{s}) \cdot t_1 \cdots t_m$ with $m \leq oa(f)$ are "protected".

[2] Leading variables in the right-hand side *also* complicate dependency pairs, but are harder to avoid; existing methods use various techniques to work around this issue.

The choice for oa is not fixed, but $oa(f) > 0$ may only hold for finitely many f. Typically, if $\sigma = (\boldsymbol{\tau}) \longrightarrow \rho_1 \to \ldots \to \rho_m \to \iota$ we choose $oa(f) = m$. We may also choose the highest m such that $f(\boldsymbol{s}) \cdot t_1 \cdots t_m$ occurs in a rule scheme. Or, in a (mostly) functional system we could choose $oa(f) = 0$ for all f; Transformations 1-2 have no effect then. A term is *limited functional* if it has the form $f(\boldsymbol{s}) \cdot t_1 \cdots t_m$ with $m < oa(f)$ (note that $f(\boldsymbol{s})$ is *not* limited functional if $oa(f) = 0$!).

Example 5. We follow the second guideline, so $oa(f)$ is the highest number m such that $f(\boldsymbol{s}) \cdot t_1 \cdots t_m$ occurs in a rule scheme. In the system $R_{\mathtt{map}}$ from Example 4 this gives output arity 0 for all symbols except `op`, which gets output arity 1.

To start, we adjust rules for the chosen output arity. For any rule $f(\boldsymbol{s}) \cdot t_1 \cdots t_n \Rightarrow r$ with $n < oa(f)$, we add a new rule $f(\boldsymbol{s}) \cdot t_1 \cdots t_n \cdot x \Rightarrow r \cdot x$. This is done because an application of the form $f(\boldsymbol{s}) \cdot \boldsymbol{t} \cdot u$ will be "protected" while $r \cdot u$ may not be.

Transformation 1 *(respecting output arity).* Given a set of rule schemes R, for every $l \Rightarrow r \in R$ with l limited functional add a rule scheme $l \cdot x \Rightarrow r \cdot x$ if this is well-typed (if $l : \alpha$ first apply the type substitution $[\alpha := \alpha \to \varepsilon]$). Repeat for all newly added rule schemes. This process terminates because $oa(f)$ is bounded, and the result, R^{res}, is finite if R is. $\Rightarrow_{R^{res}}$ and $\Rightarrow_R$ define the same relation.

Example 6. Since none of the left-hand sides of $R_{\mathtt{map}}$ are limited functional, Transformation 1 has no effect. If we had chosen, for example, $oa(\mathtt{pow}) = 2$ (this is allowed, even if there is no point in doing so), then we would have had to add four additional rules:

$$\mathtt{pow}_\sigma(F, 0) \cdot y \Rightarrow (\lambda x.x) \cdot y \qquad \mathtt{pow}_\sigma(F, \mathtt{s}(x)) \cdot y \Rightarrow \mathtt{op}(F, \mathtt{pow}(F, x)) \cdot y$$
$$\mathtt{pow}_\tau(F, 0) \cdot y \cdot z \Rightarrow (\lambda x.x) \cdot y \cdot z \qquad \mathtt{pow}_\tau(F, \mathtt{s}(x)) \cdot y \cdot z \Rightarrow \mathtt{op}(F, \mathtt{pow}(F, x)) \cdot y \cdot z$$

Here, σ is the type declaration $(\alpha \to \alpha \times \mathtt{nat}) \longrightarrow \alpha \to \alpha$ and τ is $((\alpha \to \varepsilon) \to (\alpha \to \varepsilon) \times \mathtt{nat}) \longrightarrow (\alpha \to \varepsilon) \to \alpha \to \varepsilon$.

5.2 Filling in Head Variables

With this preparation, we can proceed to a larger transformation. Let $HV(s)$ be the set of those $x_\sigma \in FVar(s)$ where x_σ occurs at the head of an application in s (so $s = C[x_\sigma \cdot t]$ for some C, t). We will replace any rule $l = C[x_\sigma \cdot t] \Rightarrow r$ by a set of rules where a limited functional term $f(\boldsymbol{y}) \cdot \boldsymbol{z}$ is substituted for x_σ.

Transformation 2 *(filling in head variables).* For every rule scheme $l \Rightarrow r$ in R^{res}, every $x_\sigma \in HV(l)$, every function symbol $f_\tau \in \mathcal{F}$ and $n < oa(f)$ such that $(\ldots) \longrightarrow \alpha_1 \to \ldots \to \alpha_n \to \sigma$ unifies with τ, let θ, χ be their most general unifiers and add a rule scheme $l\theta\delta \Rightarrow r\theta\delta$, where $\delta = [x_{\sigma\theta} := f_{\tau\chi}(\boldsymbol{y}) \cdot z_1 \cdots z_n]$ (with $y_1, \ldots, y_k, z_1, \ldots, z_n$ fresh variables). Repeat this for the newly added rule schemes. If R^{res} is finite, this process terminates and the result, R^{fill}, is also finite. Otherwise define R^{fill} as the limit of the procedure.

Example 7. Following on Example 6, the only rule with a leading variable in the left-hand side is the η-reduction rule $\lambda x_{\alpha_1}.F_{\alpha_1 \to \alpha_2}\ x_{\alpha_1} \Rightarrow F_{\alpha_1 \to \alpha_2}$. Consequently, Transformation 2 completes after one step, with a single new rule; R^{fill} contains:

$$\begin{array}{lll} \mathtt{map}(F, \mathtt{nil}) & \Rightarrow \mathtt{nil} & \mathtt{op}(F,G) \cdot x \Rightarrow F \cdot (G \cdot x) \\ \mathtt{map}(F, \mathtt{cons}(x,y)) & \Rightarrow \mathtt{cons}(F \cdot x, \mathtt{map}(F,y)) & \lambda x.F \cdot x \Rightarrow F \\ \mathtt{pow}(F, 0) & \Rightarrow \lambda x.x & \lambda x.\mathtt{op}(F,G) \cdot x \Rightarrow \mathtt{op}(F,G) \\ \mathtt{pow}(F, \mathtt{s}(x)) & \Rightarrow \mathtt{op}(F, \mathtt{exp}(F,x)) & \end{array}$$

It is not hard to see that R^{fill} and R^{res} generate the same relation. Moreover:

Lemma 2. *If $s \Rightarrow_{R^{fill}} t$ with a topmost step, then there are $l \Rightarrow r \in R^{fill}$, type substitution θ and substitution γ such that $s = l\theta\gamma$, $t = r\theta\gamma$ and $\gamma(x)$ is not limited functional for any $x \in HV(l)$.*

Proof. By definition of topmost step, there exist l, r, θ, γ such that $s = l\theta\gamma$ and $t = r\theta\gamma$; using induction on the size of $\{x_\sigma | x_\sigma \in HV(l) | \gamma(x_{\sigma\theta})$ is limited functional$\}$ and the definition of R^{fill} the Lemma follows without much effort.

5.3 Preparing Polymorphic Types

As suggested in the sketch of the transformation, we will introduce new symbols $@_\sigma$ only for those σ where it is necessary. Formally, let S be a set of functional types such that its closure under type substitution, S^c, contains all types σ where $x_\sigma \in HV(l)$ for some variable x and $l \Rightarrow r \in R^{fill}$. Transformation 4 will replace subterms $u \cdot v$ by $@(u,v)$ if $u : \tau \leq \sigma$ and u is not limited functional. There is one remaining problem: a subterm $u \cdot v$ where the type of u unifies with a type in S but is not an instance. We deal with this problem by adding some rule schemes.

Transformation 3 *(S-normalising the rules).* For every rule scheme $l \Rightarrow r \in R^{fill}$, add a rule scheme $l\theta \Rightarrow r\theta$ if either l or r has a subterm $s \cdot t$ with $s : \sigma$ not limited functional, and σ unifies with a type $\tau \in S$ such that $\tau \not\geq \sigma$. Here, θ and some χ are the most general unifiers of σ and τ. Repeat this for the newly added rules. If S and R^{fill} are both finite, this procedure terminates and the result, R^{normS}, is finite. Otherwise we define R^{normS} as the limit of the procedure.

Example 8. Consider our main Example; R^{fill} is given in Example 7. We must choose $S = \{\alpha \to \varepsilon\}$ due to the rule $\lambda x.F_{\alpha \to \varepsilon} \cdot x \Rightarrow F$. But $\alpha \to \varepsilon \geq$ any functional type, so Transformation 3 has no effect.

Again it is evident that the rewrite relations generated by R^{normS} and R^{fill} are the same. Moreover, we can derive the following (technical) result:

Lemma 3. *For $l \Rightarrow r \in R^{fill}$, type substitution θ, there are $l' \Rightarrow r' \in R^{normS}$ and type substitution χ such that $l\theta = l'\chi$, $r\theta = r'\chi$ and for $u \cdot v$ occurring in l' or r' with $u : \sigma$ not limited functional, either both $\sigma, \sigma\chi \in S^c$ or both $\sigma, \sigma\chi \notin S^c$.*

5.4 Explicit Application

Finally, then, we move on to the main substitution. For every type $\sigma \to \tau \in S$, introduce a new symbol $@_{(\sigma \to \tau \times \sigma) \longrightarrow \tau}$ and for all terms s define $\mathtt{exp}(s)$ as follows:

$$\begin{array}{ll}
\mathtt{exp}(f(s_1, \ldots, s_n)) & = f(\mathtt{exp}(s_1), \ldots, \mathtt{exp}(s_n)) \\
\mathtt{exp}(x) & = x \ (x \text{ a variable}) \\
\mathtt{exp}(\lambda x.s) & = \lambda x.\mathtt{exp}(s) \\
\mathtt{exp}(s \cdot t) & = \begin{cases} \mathtt{exp}(s) \cdot \mathtt{exp}(t) \text{ if } s \text{ limited functional or type}(s) \notin S^c \\ @(\mathtt{exp}(s), \mathtt{exp}(t)) \ \text{ otherwise} \end{cases}
\end{array}$$

That is, subterms $s \cdot t$ are replaced by $@_\sigma(s,t)$, provided the split does not occur in a "protected" functional term, and s has a "dangerous" type.

Transformation 4 *(Embedding head symbols).* Let $R^{\mathrm{noapp}} = \{\mathtt{exp}(l) \Rightarrow \mathtt{exp}(r) | l \Rightarrow r \in R^{normS}\} \cup \{@_{(\sigma \to \tau \times \sigma) \longrightarrow \tau}(x,y) \Rightarrow x \cdot y | \sigma \to \tau \in S\}$.

Transformations 1–4 preserve monomorphism and finiteness, yet R^{noapp} will not have leading (free) variables. We pose the main theorem of Section 5.

Theorem 3. *The rewrite relation $\Rightarrow_{R^{\mathrm{noapp}}}$ generated by R^{noapp} is terminating if and only if $\Rightarrow_R$ is.*

Proof (Sketch). For one direction, if $s \Rightarrow_{R^{\mathrm{noapp}}} t$ then also $s' \Rightarrow^{=}_{R^{normS}} t'$, where s', t' are s, t with occurrences of $@(u,v)$ replaced by $u \cdot v$. Equality only occurs if s has less @ symbols than t, so any infinite $\Rightarrow_{R^{\mathrm{noapp}}}$ reduction leads to an infinite $\Rightarrow_{R^{normS}}$ reduction, and R^{normS} defines the same relation as $\mathcal{R}$. For the other direction, $s \Rightarrow_{R^{res}} t$ implies $\mathtt{exp}(s) \Rightarrow^{+}_{R^{\mathrm{noapp}}} \mathtt{exp}(t)$ by induction on the size of s. For the induction step the only difficult case is when $s = u \cdot v \Rightarrow_{R^{res}} u' \cdot v$ with u limited functional while u' is not, but using Transformation 1 we can assume this is a topmost step. For the base case, if $s \Rightarrow_{R^{res}} t$ by a topmost rule step, we note that using Lemmas 2 and 3, $s = l\theta\gamma$ and $t = r\theta\gamma$ with $l \Rightarrow r \in R^{normS}$, $\gamma(x)$ not limited functional if $x \in HV(l\theta)$ and for any subterm $u \cdot v$ of l with $u : \tau$, either both τ and $\tau\theta \in S^c$ or neither. With these facts it is easy to show (using induction on the definition of $\mathtt{exp}$) that $\mathtt{exp}(l\theta\gamma) = \mathtt{exp}(l)\theta\gamma^{\mathtt{exp}}$ and $\mathtt{exp}(r)\theta\gamma^{\mathtt{exp}} \Rightarrow^{*}_{R^{\mathrm{noapp}}} \mathtt{exp}(r\theta\gamma)$. If $s \Rightarrow_\beta t$ we use that $\mathtt{exp}(u)[x := \mathtt{exp}(v)] \Rightarrow^{*}_{R^{\mathrm{noapp}}} \mathtt{exp}(u[x := v])$. Thus, any $\Rightarrow_{\mathcal{R}}$ reduction leads to a $\Rightarrow_{R^{\mathrm{noapp}}}$ reduction of at least equal length.

Example 9. Considering our example with $S = \{\alpha \to \varepsilon\}$, R^{noapp} consists of:

$$\begin{array}{llll}
\mathtt{map}(F, \mathtt{nil}) & \Rightarrow \mathtt{nil} & \mathtt{op}(F,G) \cdot x & \Rightarrow @(F, @(G,x)) \\
\mathtt{map}(F, \mathtt{cons}(x,y)) & \Rightarrow \mathtt{cons}(@(F,x), \mathtt{map}(F,y)) & \lambda x.@(F,x) & \Rightarrow F \\
\mathtt{pow}(F, 0) & \Rightarrow \lambda x.x & \lambda x.\mathtt{op}(F,G) \cdot x & \Rightarrow \mathtt{op}(F,G) \\
\mathtt{pow}(F, \mathtt{s}(x)) & \Rightarrow \mathtt{op}(F, \mathtt{exp}(F,x)) & @(F,x) & \Rightarrow F \cdot x
\end{array}$$

6 Abstractions in Left-hand Sides and β-Redexes

The next step is to get rid of rule schemes $\lambda x.l \Rightarrow r$, where an abstraction is reduced directly; rules like this will form a definite blockade to working with η-expanded terms and they make it hard to define dependency pairs. The solution is very similar to the one employed in Section 5: we identify the types of all rule schemes of this form, and replace abstractions $\lambda x.s$ of such a type σ by $\Lambda_\sigma(\lambda x.s)$, where Λ_σ is a new function symbol. As a side bonus, we will get rid of any remaining β-redexes in the rule schemes (note that the transformations of Section 5 may already have removed such redexes).

Formally, let Q be a set of types such that its closure under type substitution, Q^c, contains all types σ such that $\lambda x.l : \sigma \Rightarrow r \in R$, or $(\lambda x.s) \cdot t$ occurs in any rule scheme. We could choose the set of all such types, or for instance $\{\alpha \to \varepsilon\}$. As before we need to prepare polymorphic rule schemes for a type match.

Transformation 5 *(Q-normalising the rules).* . For every rule scheme $l \Rightarrow r \in R$, add a rule scheme $l\theta \Rightarrow r\theta$ if either l or r has a subterm $\lambda x.s : \sigma$, and σ unifies with a type $\tau \in Q$ such that $\tau \not\geq \sigma$. Here, θ and some χ are the most general unifiers of σ and τ. Repeat this for the newly added rules. If Q and R are both finite, this procedure terminates and the result, R^{normQ}, is finite. Otherwise define R^{normQ} as the limit of the procedure.

We can derive a Lemma very similar to Lemma 3, but it would not bring much news. Let us instead pass straight to the main transformation:

$$\begin{array}{ll}
\mathtt{expL}(f(s_1,\ldots,s_n)) & = f(\mathtt{expL}(s_1),\ldots,\mathtt{expL}(s_n)) \\
\mathtt{expL}(s \cdot t) & = \mathtt{expL}(s) \cdot \mathtt{expL}(t) \\
\mathtt{expL}(x) & = x \ (x \text{ a variable}) \\
\mathtt{expL}(\lambda x.s) & = \begin{cases} \Lambda_{(\sigma)\longrightarrow\sigma}(\lambda x.\mathtt{expL}(s)) & \text{if } \lambda x.s : \sigma \in Q^c \\ \lambda x.\mathtt{expL}(s) & \text{otherwise} \end{cases}
\end{array}$$

Transformation 6 *(Marking Abstractions).* $R^\Lambda := \{\mathtt{expL}(l) \Rightarrow \mathtt{expL}(r) | l \Rightarrow r \in R^{normQ}\} \cup \{\Lambda_{(\sigma)\longrightarrow\sigma}(x) \Rightarrow x | \sigma \in S\}$

It is evident that R^Λ has no rule schemes of the form $\lambda x.l \Rightarrow r$ and is β-normal. Moreover, its termination is equivalent to termination of the original system.

Theorem 4. *$\Rightarrow_{R^\Lambda}$ is terminating if and only if $\Rightarrow_R$ is.*

Proof. It is not too hard to derive that $s \Rightarrow_R t$ implies $\mathtt{expL}(s) \Rightarrow^+_{R^\Lambda} \mathtt{expL}(t)$, using that $\mathtt{expL}(C[u]) = \mathtt{expL}(C)[\mathtt{expL}(u)]$ and a separate induction for the top step (using Transformation 5 to choose the right rule and type substitution). Defining s', t' as s, t with occurrences of any Λ_σ erased, it is also obvious that $s \Rightarrow_{R^\Lambda} t$ implies $s' \Rightarrow^=_R t'$, with equality only if the former was Λ-erasing.

Example 10. Continuing the transformation of $R_{\mathtt{map}}$, we choose $Q = \{\alpha \to \varepsilon\}$ (we have no other choice, because of the rule $\lambda x.@(F_{\alpha\to\varepsilon}, x) \Rightarrow F_{\alpha\to\varepsilon}$). Transformation 5 has no effect, and Transformation 6 introduces Λ around all abstractions:

$$\begin{array}{lll}
\mathtt{map}(F, \mathtt{nil}) & \Rightarrow \mathtt{nil} & \Lambda(\lambda x.@(F,x)) \Rightarrow F \\
\mathtt{map}(F, \mathtt{cons}(x,y)) & \Rightarrow \mathtt{cons}(@(F,x), \mathtt{map}(F,y)) & \Lambda(\lambda x.\mathtt{op}(F,G)\cdot x) \Rightarrow \mathtt{op}(F,G) \\
\mathtt{pow}(F,0) & \Rightarrow \Lambda(\lambda x.x) & \Lambda_{\alpha\to\varepsilon}(F) \Rightarrow F \\
\mathtt{pow}(F,\mathtt{s}(x)) & \Rightarrow \mathtt{op}(F, \mathtt{exp}(F,x)) & @(F,x) \Rightarrow F\cdot x \\
\mathtt{op}(F,G)\cdot x & \Rightarrow @(F, @(G,x)) &
\end{array}$$

Summing Up. Combining Sections 5 and 6, we can transform a set of rule schemes, without affecting termination, to satisfy the following properties:

1. both sides of rule schemes are β-normal
2. left-hand sides l have no subterms $x \cdot s$ with x a free variable
3. left-hand sides have the form $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$ with $m \geq n$

Property (3) holds by elimination: after transforming, the left-hand side of a rule scheme is neither an abstraction, nor an application headed by an abstraction or variable. If it is a variable, $x \Rightarrow r \in R$, the AFS is non-terminating and (since termination is all we are interested in) we might replace R by the set $\{a \Rightarrow a\}$.

Henceforth, rule schemes are assumed to satisfy the requirements listed above.

7 Currying

Let us turn our eyes to the status of application. As mentioned in Section 3, an applicative AFS cannot be handled with most existing termination techniques, nor can we naively turn it into a functional system. The issues are partial application (an applicative `map` system has terms like $\mathtt{map}\cdot\mathtt{s}$ which have no functional counterpart) and leading free variables (a terminating rule $g \cdot (x \cdot 0) \Rightarrow g \cdot f(0)$ has an applicative counterpart $g \cdot (x \cdot 0) \Rightarrow g \cdot (f \cdot 0)$ which is not terminating). However, we have excluded rules with leading free variables in the left-hand side. The issue of partial application can be dealt with using η-expansion.

There are two directions we might take. Usually, we would like to *uncurry* an applicative system, transforming a term $f \cdot s \cdot t$ into $f(s,t)$. Such a form is more convenient in for instance path orderings, or to define argument filterings. On the other hand, we will have to deal with application anyway, since it is part of the term syntax; to simplify the formalism it might be a good move to *curry* terms, making the system entirely applicative.

Transformation 7 *(Currying).* Let R be a set of rules schemes over a set of function symbols $\mathcal{F}$. We define the following mapping on type declarations: $\mathtt{flat}((\sigma_1 \times \ldots \sigma_n) \longrightarrow \tau) = \sigma_1 \to \ldots \to \sigma_n \to \tau$. Next we define the mapping `flat` from functional terms over $\mathcal{F}$ to applicative terms over the 'flattened version' of $\mathcal{F}$, notation $\mathcal{F}^{\mathtt{flat}}$, as follows:

$$\begin{array}{ll}
\mathtt{flat}(f_\sigma(s_1,\ldots,s_n)) & = f_{\mathtt{flat}(\sigma)} \cdot \mathtt{flat}(s_1) \cdots \mathtt{flat}(s_n) \\
\mathtt{flat}(\lambda x.s) & = \lambda x.\mathtt{flat}(s) \\
\mathtt{flat}(s \cdot t) & = \mathtt{flat}(s) \cdot \mathtt{flat}(t) \\
\mathtt{flat}(x) & = x \quad (x \text{ a variable})
\end{array}$$

The flattened version $R^{\mathtt{flat}}$ of the set of rule schemes R consists of the rule scheme $\mathtt{flat}(l) \Rightarrow \mathtt{flat}(r)$ for every rule scheme $l \Rightarrow r$ in R.

Theorem 5. *$\Rightarrow_R$ is well-founded on terms over $\mathcal{F}$ if and only if $\Rightarrow_{R^{\mathtt{flat}}}$ is well-founded on terms over $\mathcal{F}^{\mathtt{flat}}$.*

Proof. It is easy to see that $s \Rightarrow_R t$ implies $\mathtt{flat}(s) \Rightarrow_{R^{\mathtt{flat}}} \mathtt{flat}(t)$ (with induction on the size of s, and a separate induction for topmost steps to see that flattening is preserved under substitution); this provides one direction. For the other, let $\mathtt{flat}^{-1}$ be the "inverse" transformation of $\mathtt{flat}$, which maps occurrences of $f \cdot s_1 \cdots s_k$ with $f_{(\sigma_1 \times \ldots \times \sigma_n) \longrightarrow \tau} \in \mathcal{F}$ to $\lambda x_{k+1} \ldots x_n.f(\mathtt{flat}^{-1}(s_1),\ldots,\mathtt{flat}^{-1}(s_k), x_{k+1},\ldots,x_n)$ if $k < n$ or to $f(\mathtt{flat}^{-1}(s_1),\ldots,\mathtt{flat}^{-1}(s_n)) \cdot \mathtt{flat}^{-1}(s_{n+1}) \cdots \mathtt{flat}^{-1}(s_k)$ otherwise. It is not hard to see that $\mathtt{flat}^{-1}(s)[\boldsymbol{x} := \mathtt{flat}^{-1}(\boldsymbol{t})] \Rightarrow_\beta^* \mathtt{flat}^{-1}(s[\boldsymbol{x} := \boldsymbol{t}])$, and this $\Rightarrow_\beta^*$ is an equality if $HV(s) = \emptyset$. Therefore, and because $\mathtt{flat}^{-1}(R^{\mathtt{flat}})$ is exactly R, $\mathtt{flat}^{-1}(s) \Rightarrow_R^+ \mathtt{flat}^{-1}(t)$ holds if $s \Rightarrow_{R^{\mathtt{flat}}} t$.

Note the *if and only if* in Theorem 5. Because of this equivalence the theorem works in two ways. We can turn a functional system applicative, but also turn an applicative system functional, simply by taking the inverse of Transformation 7. For an applicative system, there are usually many sets of corresponding functional rules, all of which are equivalent for the purpose of termination.

Example 11. Our running example can be transformed into the applicative AFS:

$$\begin{array}{llll}
\mathtt{pow} \cdot F \cdot 0 & \Rightarrow \Lambda \cdot (\lambda x.x) & \Lambda \cdot (\lambda x.@ \cdot F \cdot x) & \Rightarrow F \\
\mathtt{pow} \cdot F \cdot (\mathtt{s} \cdot x) & \Rightarrow \mathtt{op} \cdot F \cdot (\mathtt{exp} \cdot F \cdot x) & \Lambda \cdot (\lambda x.\mathtt{op} \cdot F \cdot G \cdot x) & \Rightarrow \mathtt{op} \cdot F \cdot G \\
\mathtt{op} \cdot F \cdot G \cdot x & \Rightarrow @ \cdot F \cdot (@ \cdot G \cdot x) & @ \cdot F \cdot x & \Rightarrow F \cdot x \\
\mathtt{map} \cdot F \cdot \mathtt{nil} & \Rightarrow \mathtt{nil} & \Lambda \cdot F & \Rightarrow F \\
\mathtt{map} \cdot F \cdot (\mathtt{cons} \cdot x \cdot y) & \Rightarrow \mathtt{cons} \cdot (@ \cdot F \cdot x) \cdot (\mathtt{map} \cdot F \cdot y) & &
\end{array}$$

Related Work. In first-order rewriting, the question whether properties such as confluence and termination are preserved under currying or uncurrying is studied in [5,6,2]. In [6] a currying transformation from (functional) term rewriting systems (TRSs) into applicative term rewriting systems (ATRSs) is defined; a TRS is terminating if and only if its curried form is. In [2], an uncurrying transformation from ATRSs to TRSs is defined that can deal with partial application and leading variables, as long as they do not occur in the left-hand side of rewrite rules. This transformation is sound and complete with respect to termination.

However, these results do not apply to AFSs, both due to the presence of typing and because AFSs use a mixture of functional and applicative notation. We may for instance have terms of the form $f(x) \cdot y$, and currying might introduce new interactions via application.

8 η-Expansion

Finally, we consider η-expansion. It would often be convenient if we could assume that every term of some functional type $\sigma \to \tau$ has the form $\lambda x_\sigma.s$, which only reduces if its subterm s does. This is the case if we work modulo η, equating $s : \sigma \to \tau$, with s not an abstraction, to $\lambda x_\sigma.(s \cdot x_\sigma)$. As is well-known, simply working modulo η in the presence of β-reduction causes problems. Instead, we will limit reasoning to η-long terms.

Theorem 6. *Let $\mathcal{R}$ be a set of rules in restricted η-long form, that is, $l = l\uparrow^\eta_{FVar(l)}$ and $r = r\uparrow^\eta_{FVar(r)}$ for every rewrite rule $l \Rightarrow r$ in $\mathcal{R}$. Then the set of η-long terms is closed under rewriting. Moreover, the rewrite relation $\Rightarrow_\mathcal{R}$ is terminating on terms iff it is terminating on η-long terms.*

Proof. Evidently, if $\Rightarrow_\mathcal{R}$ is terminating then it is terminating on all η-long terms. For the less obvious direction, we see that $s \Rightarrow_\mathcal{R} t$ implies $s\uparrow^\eta \Rightarrow^+_\mathcal{R} t\uparrow^\eta$. Hence any infinite reduction can be transformed to an infinite reduction on η-long terms. Writing $\gamma^\uparrow := \{x \mapsto \gamma(x)\uparrow^\eta | x \in \mathrm{dom}(\gamma)\}$, a simple inductive reasoning shows that $s\uparrow^\eta_V \gamma^\uparrow \Rightarrow^*_\beta s\gamma\uparrow^\eta$ if $V = \mathrm{dom}(\gamma)$, and this is an equality if $HV(s) = \emptyset$. Thus, if $s \Rightarrow_\mathcal{R} t$ by a topmost reduction, then also $s\uparrow^\eta = l\gamma\uparrow^\eta = l\uparrow^\eta_{FVar(l)}\gamma^\uparrow = l\gamma^\uparrow \Rightarrow_\mathcal{R} r\gamma^\uparrow = r\uparrow^\eta_{FVar(r)}\gamma^\uparrow \Rightarrow_\beta r\gamma\uparrow^\eta = t\uparrow^\eta$. This forms the base case for an induction on s, which proves $s\uparrow^\eta \Rightarrow^+_\mathcal{R} t\uparrow^\eta$ whenever $s \Rightarrow_\mathcal{R} t$.

The requirement that the rules should be η-long is essential. Consider for example the system with a single rule $f_{o\to o} \cdot x_o \Rightarrow g_{(o\to o)\to o} \cdot f_{o\to o}$. The relation generated by this rule is terminating, but evidently the set of η-long terms is not closed under rewriting. The η-long variation of this rule, $f_{o\to o} \cdot x_o \Rightarrow g_{(o\to o)\to o} \cdot (\lambda y_o.f_{o\to o} \cdot y_o)$, is not terminating, as the left-hand side can be embedded in the right-hand side. This example is contrived, but it shows that we cannot be careless with η-expansion. However, when developing methods to prove termination of a system the most essential part of any transformation is to preserve *non-termination*. At the price of completeness, we can use Transformation 8:

Transformation 8 *(η-expanding rules).* Let $\mathcal{R}$ be a set of rules. Define $\mathcal{R}^\uparrow$ to be the set consisting of the rules $(l \cdot x^1_{\sigma_1} \cdots x^n_{\sigma_n})\uparrow^\eta_V \Rightarrow (r \cdot x^1_{\sigma_1} \cdots x^n_{\sigma_n})\uparrow^\eta_V$, for every rule $l \Rightarrow r$ in $\mathcal{R}$, with $l : \sigma_1 \to \ldots \sigma_n \to \iota$, all $x^i_{\sigma_i}$s fresh variables, and $V = FVar(l) \cup \{x^1_{\sigma_1}, \ldots, x^n_{\sigma_n}\}$.

The proof of the following theorem is a straightforward adaptation of the proof of Theorem 6.

Theorem 7. *If the rewrite relation generated by $\mathcal{R}^\uparrow$ is terminating on η-long terms, then the relation generated by $\mathcal{R}$ is terminating on the set of terms.*

Note that Theorems 6 and 7 concern *rules*, not rule schemes. The η-expansion of a terminating set of rule schemes may not be terminating, as demonstrated

by the system with $R = \{f_{\alpha\to\alpha} \cdot g_\alpha \Rightarrow h_\alpha,\ h_{\mathtt{nat}\to\mathtt{nat}} \cdot 0_{\mathtt{nat}} \Rightarrow f_{(\mathtt{nat}\to\mathtt{nat})\to\mathtt{nat}\to\mathtt{nat}} \cdot g_{\mathtt{nat}\to\mathtt{nat}} \cdot 0_{\mathtt{nat}}\}$. Thus, η-expansion is mainly useful on monomorphic systems, or for termination methods which, given rule schemes R, prove termination of $\mathcal{R}_R^{\uparrow}$.

9 Conclusions

We have seen various techniques to transform AFSs, essentially making it possible to pose restrictions on terms and rule schemes without losing generality. Considering existing results, this has various applications:

Applicative terms As mentioned before, most applicative systems cannot be dealt with directly. Consider for example the system with symbols $\mathtt{split}_{\mathtt{nat}\to\mathtt{tuple}}$ and $\mathtt{pair}_{\alpha\to\varepsilon\to\mathtt{tuple}}$ which has the following rule:

$$\mathtt{split} \cdot (x_{\mathtt{nat}\to\mathtt{nat}} \cdot y_{\mathtt{nat}}) \Rightarrow \mathtt{pair} \cdot x_{\mathtt{nat}\to\mathtt{nat}} \cdot y_{\mathtt{nat}}$$

Even the computability path ordering [1], which is the latest definition in a strengthening line of path orderings, cannot deal with this rule. However, using Transformations 1–4 we introduce $@_{(\mathtt{nat}\to\mathtt{nat}\times\mathtt{nat})\longrightarrow\mathtt{nat}}$ and the system becomes:

$$\mathtt{split} \cdot @(x,y) \Rightarrow \mathtt{pair} \cdot x \cdot y \qquad @(x,y) \Rightarrow x \cdot y$$

This system has the same curried form as:

$$\mathtt{split}(@(x,y)) \Rightarrow \mathtt{pair}(x,y) \qquad @(x,y) \Rightarrow x \cdot y$$

Consequently, termination of one implies termination of the other by Theorem 5. But the latter is immediate with HORPO [4], using a precedence $@ >_{\mathcal{F}} \mathtt{pair}$.

CPO The latest recursive path ordering, CPO, is defined only for monotonic systems where all symbols have a data type as output type. It cannot, for instance, deal with a system with rules:

$$\begin{aligned} \mathtt{emap}(F, \mathtt{nil}) &\Rightarrow \mathtt{nil} \\ \mathtt{emap}(F, \mathtt{cons}(x,y)) &\Rightarrow \mathtt{cons}(F \cdot x, \mathtt{emap}(\mathtt{twice}(F), y)) \\ \mathtt{twice}(F) \cdot x &\Rightarrow F \cdot (F \cdot x) \end{aligned}$$

Here, $\mathtt{twice}$ has type declaration $(\mathtt{nat}\to\mathtt{nat}) \longrightarrow \mathtt{nat}\to\mathtt{nat}$. By Theorem 6 we can η-expand these rules, the result of which has the same curried form as:

$$\begin{aligned} \mathtt{emap}(F, \mathtt{nil}) &\Rightarrow \mathtt{nil} \\ \mathtt{emap}(F, \mathtt{cons}(x,y)) &\Rightarrow \mathtt{cons}(F \cdot x, \mathtt{emap}(\lambda z.\mathtt{twice}(F,z), y)) \\ \mathtt{twice}(F,x) &\Rightarrow F \cdot (F \cdot x) \end{aligned}$$

Thus, if this system can be proved terminating with CPO (which it can, if a reverse lexicographical ordering is used), the original system is terminating. CPO can be applied on any monomorphic system in this way, although the transformation may lose termination due to the η-expansion.

Dependency Pairs Since rules can be assumed to have a form $f(l_1,\ldots,l_n) \cdot l_{n+1} \cdots l_m$, the dependency pair method for AFSs in [8] is now applicable without restrictions other than monotonicity; a transformation tool which has Transformations 1–6 built into the input module could build around a dependency pair framework without losing out.

Summary and Future Work. In this paper we discussed transformations which simplify Algebraic Functional Systems significantly. We saw that polymorphism only has a function in defining rule schemes, that rule schemes can be assumed to be β-normal and that there is no need for leading free variables in the left-hand side of rules. We know that applicative and functional notation can be interchanged, and rule schemes can be assumed to have a form $f \cdot l_1 \cdots l_n \Rightarrow r$ with f a function symbol. Moreover, when we are interested only in proving termination, we may η-expand the rules and restrict attention to η-long terms.

A monomorphic version of the transformations given here was implemented in WANDA v1.0 [7], which participated in the Termination Competition 2010 [13].

In the future, we intend to look further into other formalisms, and give conditions and techniques to transfer results across.

Acknowledgement

We are indepted to the anonymous referees for their remarks which helped to improve the paper.

References

1. Blanqui, F., Jouannaud, J.-P., Rubio, A.: The computability path ordering: The end of a quest. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 1–14. Springer, Heidelberg (2008)
2. Hirokawa, N., Middeldorp, A., Zankl, H.: Uncurrying for termination. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 667–681. Springer, Heidelberg (2008)
3. Jouannaud, J.-P., Okada, M.: A computation model for executable higher-order algebraic specification languages. In: 6th IEEE Symposium on Logic In Computer Science, pp. 350–361. IEEE Press, Amsterdam (1991)
4. Jouannaud, J.-P., Rubio, A.: The higher-order recursive path ordering. In: 14th IEEE Symposium on Logic In Computer Science, pp. 402–411. IEEE Press, Los Alamitos (1999)
5. Kahrs, S.: Confluence of curried term-rewriting systems. Journal of Symbolic Computing 19, 601–623 (1995)
6. Kennaway, R., Klop, J.W., Sleep, M.R., de Vries, F.J.: Comparing curried and uncurried rewriting. Journal of Symbolic Computing 21(1), 15–39 (1996)
7. Kop, C.: Wanda, `http://www.few.vu.nl/~kop/code.html`
8. Kop, C., van Raamsdonk, F.: Higher order dependency pairs for algebraic functional systems. In: RTA 2011 (to appear, 2011)
9. Kusakari, K., Isogai, Y., Sakai, M., Blanqui, F.: Static dependency pair method based on strong computability for higher-order rewrite systems. IEICE Transactions on Information and Systems 92, 2007–2015 (2009)
10. Nipkow, T.: Higher-order critical pairs. In: 6th IEEE Symposium on Logic in Computer Science, pp. 342–349. IEEE Press, Amsterdam (1991)
11. van de Pol, J.C.: Termination of Higher-order Rewite Systems. PhD thesis, University of Utrecht (1996)
12. Sakai, M., Watanabe, Y., Sakabe, T.: An extension of the dependency pair method for proving termination of higher-order rewrite systems. IEICE Transactions on Information and Systems E84-D(8), 1025–1032 (2001)
13. Termination Portal (Wiki), `http://www.termination-portal.org/`